# Early Experiences With The OpenMP Accelerator Model

C. Liao, Y. Yan, B. de Supinsky, D. Quinlan, B. Chapman

May 13, 2013

International Workshop on OpenMP (IWOMP) 2013
Canberra, Australia
September 16, 2013 through September 18, 2013

## Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

# Early Experiences With The OpenMP Accelerator Model

Chunhua Liao[1], Yonghong Yan[2], Bronis R. de Supinski[1], Daniel J. Quinlan[1]
and Barbara Chapman[2]

[1] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
{liao6,dquinlan,desupinski1}@llnl.gov
[2] Department of Computer Science, University of Houston
{yanyh,chapman}@cs.uh.edu

**Abstract.** A recent trend in mainstream computer nodes is the combined use of general-purpose multicore processors and specialized accelerators such as GPUs and DSPs in order to achieve better performance and to reduce power consumption. To support this trend, the OpenMP Language Committee has approved a set of extensions to OpenMP (referred to as the OpenMP accelerator model). The initial version is the subject of Technical Report 1 (TR1) while OpenMP 4.0 Release Candidate 2 (RC2) further refines the extensions.
In this paper, we examine the newly released accelerator directives and create an initial reference implementation, referred to as HOMP (Heterogeneous OpenMP). Focused on targeting NVIDIA GPUs, our work is based on an existing OpenMP implementation in the ROSE source-to-source compiler infrastructure. HOMP includes extensions to parse the new constructs and to represent them in the AST and other compiler translation details. Further we provide initial runtime support. For our evaluation, we have adapted a few existing OpenMP codes to use the accelerator model directives and present preliminary performance results. Finally, we critique the accelerator model in terms of its impact on developers and compiler writers and suggest possible improvements.

## 1 Introduction

Heterogeneous computer architectures that combine general-purpose multicore CPUs with specialized accelerators have become a viable solution to build high performance supercomputers, as demonstrated by Titan at ORNL (NVIDIA GPGPUs) and Stampede at TACC (Intel Xeon Phi) in the recent top500 list. Multicore CPUs are good at processing coarse-grained, irregular tasks; while accelerators excel in certain workloads such as large-scale data parallel and finer-grained vector processing. However, to exploit their computation capabilities

LLNL-CONF-636479. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was also supported by the National Science Foundations Computer Research Infrastructure program under Award No. CNS-1205708.

efficiently has required significant programmer effort to optimize an application program with respect to the specific hardware features of each type of accelerator.

Programming models such as OpenCL, CUDA and Brook provide mechanisms for an application to exploit the hardware capabilities of accelerators. High-level programming models, such as OpenACC [1], aim to provide an easier migration option from a sequential or parallel CPU version to the use of accelerators, typically GPGPUs. However, using these programming models to exploit their capabilities completely still poses significant challenges, even for expert programmers. Using multiple programming models in one application, as is likely with models that provide accelerator support that is distinct from CPU models, increases code complexity and decreases its portability. Mixing multiple programming models also complicates the compiler and runtime support due to the language complexity and to support runtime interoperability.

OpenMP has proven to be a productive solution for parallel programming with CPUs in shared memory systems. Recently, the OpenMP Language Committee has been working toward a single specification that supports heterogeneous computation nodes using both CPUs and accelerators. The committee has developed a set of extensions that they released first as a dedicated Technical Report 1 (TR1) and then as part of OpenMP 4.0 Release Candidate 2 (RC2) [2]. The extensions in this OpenMP accelerator model build on existing OpenMP concepts and constructs to provide a unified model for GPUs and CPUs. This model relies on compiler analysis and transformations to generate code that can execute on accelerators for specified source code regions, as well as runtime support to provide data movement and other support for hybrid execution.

In this paper, we review the OpenMP accelerator model and share our experiences of creating an initial implementation, the Heterogeneous OpenMP (HOMP) compiler. We have two goals: to provide early feedback on the usability of the OpenMP accelerator model and its impact on compiler writers; and to create a reference implementation for the extensions that the research community can leverage to explore further extensions.

The rest of the paper is organized as follow. Section 2 reviews the major accelerator extensions to OpenMP. Section 3 describes our initial implementation of those extensions. We present our preliminary results in Section 4 and critique the current model in Section 5. Section 6 presents related work. Section 7 concludes the paper and discusses our future work.

## 2 The OpenMP Accelerator Model

OpenMP 4.0 Release Candidate 2 [2] extends the execution model of the specification to support accelerators with device constructs. The OpenMP accelerator model assumes that a computation node has a host device connected with one or multiple accelerators as target devices. It uses a host-centric model in which a host device “offloads” code regions and data to accelerators for execution, specified using the `target` construct. This construct causes data and an executable to be copied (offloaded) to the accelerator before computation.

The OpenMP memory model is extended so that the code region has its own data environment. A device appears to have an independent shared memory, although copies cannot be assumed. Data-mapping attributes, specified using the map clause, define how variables are handled for the device data environments, including allocation, initialization and assignment to the host variables at the end of a target [data] region.

A device, which can be any logical execution engine defined by an implementation, has threads that behave almost the same as threads on the host device. Initially, only a single thread starts on a device to run an implicit task region. This single thread can fork more threads later when it encounters parallel constructs. It can also generate tasks as can its CPU counterpart. RC2 also introduced "thread teams" for organizing device threads in a structured way, which we will discuss in more detail in later sections.

### 2.1 Directives for Data and Computation Offloading

The target directive is introduced to offload data and computation to a device. It can have clauses to indicate a target device (device), data-mapping attributes (map), and an if condition to control the use of offloading at runtime.

The device data environment reflects the data-mapping attributes specified by the map clauses and the existing device data environment, which may have previously mapped variables due to target data constructs. Data mapping attributes include alloc, to, from, and tofrom, which determine how the list item is allocated, initialized and copied (handled) at region completion. The map clause can apply to "array sections", which designate a subset of an array, building on standard Fortran syntax or syntax added to OpenMP to support the concept for C and C++ for native arrays and pointer-based arrays.

Figure 1 shows a Jacobi iteration kernel written using the OpenMP 4.0 RC2 specification. One directive (line 6 and 7 of Figure 1) converts the existing host OpenMP code to device code. Since a target region can run on a host device whenever an implementation chooses, programmers should generally write a host version before adding accelerator-specific directives.

To avoid repetitive creation and cancellation of device data environments, the target data directive defines a device data region, in which multiple target regions can share the same device data. As shown at lines 1 and 2 in Figure 1, a device data region is defined before the while loop that contains the kernel. So each kernel launch within the while loop can reuse the enclosing data region. However, the map type of a data item in a map clause of a target construct can change if it is enclosed in a data region. For example, the map type for uold is to (copy the new values generated to the device) at line 6 while it has an alloc type at line 2. The reason is that outside of the while loop, uold is neither live-in nor live-out. Users must use care for their choice of map type depending on where they define data regions.

Another new directive, target update, can have motion clauses (to and from) and a device clause. According to the motion clauses, this construct makes a set of variables in the device data environment consistent with their original

```
#pragma omp target data map(to:n, m, omega, ax, ay, b, f[0:n][0:m]) \
          map(tofrom:u[0:n][0:m]) map(alloc:uold[0:n][0:m])
while ((k<=mits)&&(error>tol))
{
// a ”target + parallel for” loop copying u to uold is omitted ...
#pragma omp target map(to:n, m, omega, ax, ay, b, f[0:n][0:m], \
                   uold[0:n][0:m]) map(tofrom:u[0:n][0:m])
#pragma omp parallel for private(resid,j,i) reduction(+:error)
for (i=1;i<(n-1);i++)
  for (j=1;j<(m-1);j++)
  {
    resid = (ax*(uold[i-1][j] + uold[i+1][j])\
        + ay*(uold[i][j-1] + uold[i][j+1])+ b * uold[i][j] - f[i][j])/b;
    u[i][j] = uold[i][j] - omega * resid;
    error = error + resid*resid ;
  } // the rest code omitted ...
}
```

**Fig. 1.** Jacobi kernel using accelerator directives

list items. With it, programmers can selectively update data values between the host and device data environments. Another directive, declare target, specifies that variables, functions (C, C++ and Fortran), and subroutines (Fortran) are mapped to (compiled for) a device. This directive generates device binaries for code that is not in the lexical scope of the target region, including the use of OpenMP constructs.

### 2.2 Directives for Thread Hierarchy

Accelerators are often massively parallel architecture devices that support hundreds or even thousands of concurrent threads with a hierarchical organization. Language constructs that allow users to manage the thread hierarchy are often needed. For example, CUDA provides the hierarchy of threads in blocks and grids. RC2 provides the teams and distribute constructs to manage a two-level thread hierarchy. Previously, OpenMP included the concept of a thread team, a group of synchronizable threads, to support nested parallelism. The teams construct creates a league or group of these thread teams. Initially each team in the league has one thread; subsequent parallel regions can create more threads in that team. The distribute construct specifies that the iterations of an associated loop are distributed across the master threads of all teams that execute the teams region to which the distribute region binds. Figure 2 gives a simple example of calculating the sum of an integer array using these constructs. The complex semantics lead to less intuitive code than existing OpenMP constructs. Without combined constructs, users must manually split a single loop into two loops in order to schedule the original loop at two levels of threads. The resulting code may be only useful for certain accelerators types such as NVIDIA GPGPUs.

## 3 HOMP: A Prototype Implementation

We are building a prototype implementation (referred to as HOMP, short for Heterogeneous OpenMP) for the OpenMP accelerator model. The current fo-

```
int sum = 0;
int A[1000];
...
#pragma omp target map(to: A[0:1000])
#pragma omp teams num_teams(2) num_threads(100) reduction(+:sum)
#pragma omp distribute
  for (i_0 = 0; i_0 < 1000; i_0 += 500)
#pragma omp parallel for reduction(+:sum)
  for (i = i_0; i < i_0 + 500; i++)
    sum += A[i];
```

**Fig. 2.** Calculating sum explicitly using multiple contention teams

cus is to generate CUDA code because of the popularity of NVIDIA GPUs for high performance computing. Built upon ROSE's OpenMP implementation [3], HOMP is designed as an open implementation that the community can leverage to explore the design space of OpenMP extensions for accelerators. In particular, we have extended ROSE's pragma parsing to parse the new directives and clauses. We added new node types to ROSE's intermediate representation for the new directives and clauses related to accelerators, including the target and target data regions and the map clause. We similarly extended OpenMP lowering and runtime support. We give more details about the fundamental OpenMP implementation and our additional work for device constructs below.

### 3.1 ROSE and HOMP

HOMP is built on ROSE [4], a source-to-source compiler infrastructure developed at Lawrence Livermore National Laboratory to build compilers or program transformation and analysis tools for large-scale C/C++ and Fortran applications. Essentially, ROSE provides an object-oriented abstract syntax tree (AST) with a set of parsing, unparsing, analysis and transformation interfaces allowing users to build translators, analyzers, optimizers, and specialized tools quickly.

The existing ROSE OpenMP implementation [3] supports OpenMP 3.0 directives for C, C++ and a subset of Fortran. Internally, ROSE's OpenMP support works through the following steps: 1) AST generation of input code. 2) OpenMP pragma parsing since the frontends used by ROSE do not recognize OpenMP. 3) AST patching for adding new nodes and edges representing OpenMP directives and clauses. These new OpenMP-specific AST nodes are created to represent the semantics of OpenMP intuitively. For example, a node named (SgOmpParallelStatement) with a body statement block represents an **omp parallel** region. 4) OpenMP lowering to generate multithreaded code calling runtime functions. 5) Generate (unparse) transformed source code from the AST. A backend compiler will be transparently invoked to generate object code from the output code. 6) Link with runtime support to generate the executable. ROSE defines a generic runtime layer (XOMP) that abstracts common runtime support for OpenMP implementations and insulates the compiler translation from minor changes to runtime libraries. As a result, ROSE is unique in that a single set of OpenMP translations can work with multiple OpenMP runtime libraries.

### 3.2 Implementing the Accelerator Model

*Target Regions* A target region starts a sequential execution of the initial implicit task on a target device. Using the latest CUDA 5.0 environment and GPUs with Compute Capability 3.5 or beyond, a target region can be implemented with a kernel launch configured with a single thread block with a single thread. When a parallel or teams region is encountered, dynamic parallelism can be used to launch another CUDA kernel configured with the requested number of thread blocks and threads per block.

We have at least two choices for CUDA environments that lack support for dynamic parallelism. The first one is to launch enough thread blocks and threads per block when the first sequential region of a target region is encountered despite actually using only a single thread. However, accurate estimation of the thread and block counts is difficult since later parallel regions may occur in functions and may dynamically change the counts. This choice may also waste energy if the sequential region has a long duration. The other choice translates each sequential portion and parallel portion into an independent kernel launch, with unnecessary synchronization after each launch. We consider this the better choice.

For a target region immediately followed by a parallel region, directly launching a multiple-thread execution kernel without an initial sequential part is the best choice. This choice more intuitively fits the semantics that users often express for GPUs. Thus, combined omp target parallel or omp target teams parallel are more useful and more intuitive than their separate forms.

*Parallel Regions and Teams* With this target region implementation, each enclosed parallel region can be implemented as a separate kernel launch. However, with CUDA, only threads within the same thread block can (easily) synchronize. Unless developers explicitly use teams with parallel, an implementation cannot blindly spawn threads across multiple blocks since the parallel region may have synchronization points in the middle of its execution.

When the programmer does not specify the teams construct, compilers can limit the spawned threads to be those belonging to a single thread block, without leveraging all available GPU threads. Alternatively, they can use analysis to rule out synchronization points in the middle of the parallel region and then freely spawn threads across thread blocks as needed. This optimization requires a scan of the parallel region for synchronization constructs in the middle, such as barrier and atomic, and any unresolvable function calls that might contain such constructs. This alternative allows more parallelism in exchange for increased compiler complexity. Another solution would introduce a new clause such as no-middle-sync for a parallel region to indicate explicitly that the region does not contain any synchronization points.

For example, we outline the source code of the parallel region in Figure 1 so it can be transformed into the CUDA kernel in Figure 3. We insert CUDA execution configuration and kernel launch statements at lines 19 to 27 in Figure 4. Two runtime functions xomp_get_maxThreadsPerBlock() and xomp_get_max1DBlock() obtain the default execution configuration based on the hardware information and

```
__global__ void OUT__1__10117__(int n, int m, float omega, float ax, \
                       float ay, float b, float *_dev_per_block_error , \
                       float *_dev_u , float *_dev_f , float *_dev_uold)
{
  /* local variables for  loop , reduction, etc */
  int _p_j; float _p_error; _p_error = 0; float _p_resid;
  int _dev_i , _dev_lower , _dev_upper;

  /* Obtain loop bounds for current thread of current block */
  XOMP_accelerator_loop_default (1, n-2, 1, &_dev_lower , &_dev_upper);
  for (_dev_i = _dev_lower; _dev_i<= _dev_upper; _dev_i ++) {
    for (_p_j = 1; _p_j < (m - 1); _p_j++) {
/*  replace original variables with device variables
    linearize 2-D array accesses */
      _p_resid = (((((ax * (_dev_uold[(_dev_i - 1) * 512 + _p_j]  \
                             + _dev_uold[(_dev_i + 1) * 512 + _p_j])) \
                       + (ay * (_dev_uold[_dev_i * 512 + (_p_j - 1)] \
                       + _dev_uold[_dev_i * 512 + (_p_j + 1)])))     \
                       + (b * _dev_uold[_dev_i * 512 + _p_j]))       \
                       - _dev_f[_dev_i * 512 + _p_j]) / b);
     _dev_u[_dev_i * 512 + _p_j] = (_dev_uold[_dev_i * 512 + _p_j] \
                                  - (omega * _p_resid));
     _p_error = (_p_error + (_p_resid * _p_resid));
    }
  }
  /* thread block level reduction for float type*/
  xomp_inner_block_reduction_float(_p_error,_dev_per_block_error,6);
}
```

**Fig. 3.** Generated CUDA kernel

the number of iterations. Our current strategy uses the full number of supported hardware threads within a thread block before using more blocks.

*Data Handling* Based on the specified map types, the map clause guides the translation of device variable declarations, memory allocation, value copying between CPU memory and GPU memory, and deallocation. Since a variable in a nested map clause may already exist in an enclosing data environment (e.g., array u shown at both lines 2 and 7 in Figure 1), an implementation must track active device data environments to reuse the versions in enclosing data environments when they exist.

We track the data environments that target data and target constructs create in a stack and add runtime functions for that purpose. First, xomp_DDE_Enter() (line 2 in Figure 4; DDE stands for deviceDataEnvironment) initializes a data structure for each new data environment and pushes it onto the stack. The data structure stores information about variables allocated within the current data environment. Second, xomp_DDE_GetInheritedVariable() (line 6) checks if a variable in a map clause already exists in enclosing environments. Third, xomp_DDE_AddVariable (line 13) registers a newly mapped variable with its original address, device address, size, and a copy back flag. Finally, based on stored information for mapped variables, xomp_DDE_Exit() (line 35) transparently copies data back to the host and deallocates device memory deallocation before it is popped from the stack.

We linearize the storage of array variables with two or more dimensions. Accordingly, we replace all references to the original array elements with ref-

```
1  /* Initialize a new data environment, push it to a stack */
2  xomp_deviceDataEnvironmentEnter();
3
4  int _dev_u_size = sizeof(float ) * (n - 0) * (m - 0);
5  /* Try to grab a mapped variable from enclosing data environments */
6  float *_dev_u=(float*)xomp_deviceDataEnvironmentGetInheritedVariable \
7                ((void*)u, _dev_u_size);
8  /* If not inheritable, allocate and register the mapped variable */
9  if (_dev_u == NULL)
10 {
11   _dev_u = ((float *)(xomp_deviceMalloc(_dev_u_size)));
12 /* Register CPU address, device address, size, and a copy-back flag */
13   xomp_deviceDataEnvironmentAddVariable ((void*)u, _dev_u_size, \
14                                         (void*) _dev_u, true);
15   // data copy from Host to Device also here if specified
16 }
17 ... // handling of other variables is omitted
18
19 /* Execution configuration: threads per block and total block numbers*/
20 int _threads_per_block_ = xomp_get_maxThreadsPerBlock();
21 int _num_blocks_ = xomp_get_max1DBlock((n - 1) - 1);
22 float *_dev_per_block_error = (float *)(xomp_deviceMalloc( \
23                                       _num_blocks_ * sizeof(float)));
24 /* Launch the CUDA kernel ... */
25 OUT__1__10117__<<<_num_blocks_, _threads_per_block_, \
26                  (_threads_per_block_ * sizeof(float))>>> \
27     (n,m,omega,ax,ay,b,_dev_per_block_error,_dev_u,_dev_f,_dev_uold);
28 /*  Beyond thread block reduction */
29 error = xomp_beyond_block_reduction_float(_dev_per_block_error, \
30                                         _num_blocks_,6);
31 /*  Data deallocation, copy-back, etc.  */
32 xomp_freeDevice(_dev_per_block_error);
33 ...
34 /*Copy back and deallocate variables within this environment, pop stack */
35 xomp_deviceDataEnvironmentExit();
```

**Fig. 4.** Generated kernel configuration and launch code

erences that use the device variable with a linear address calculation (e.g., _dev_u[_dev_i*512 + _p_j] at line 21 in Figure 3).

For simplicity, we use a two-level algorithm to implement reductions that leverage GPUs. One level is within each CUDA thread block and the other is across multiple thread blocks on the host side. We provide a set of runtime functions (e.g., xomp_inner_block_reduction_*() and xomp_beyond_block_reduction_*()) to support these two-level reductions. Figure 3 shows example code for the GPU-based inner-block reduction (line 27). The CPU side's across-block reduction is shown at line 29 of Figure 4.

*Loop and Distribute Directives* By default, only the outermost loop is affected by the loop constructs unless a collapse clause is specified to allow an implementation to combine multiple loops into a larger iteration space. Three choices to schedule loop iterations among GPU threads are available: 1) use only master threads of multiple thread blocks when distribute is used right before the loop; 2) use threads from a single thread block; or 3) use a combination of multiple blocks and multiple threads per block, when applicable. Figure 3 shows an example translation. The translation calls XOMP_accelerator_loop_default() at line 10 to obtain the bounds for the current thread within the current thread block. No loop splitting is needed even for scheduling loops across teams.

On a final note, all runtime functions are designed to have a C binding so that they can interoperate easily with multiple programming languages including C, C++ and Fortran. The function interfaces are designed to be similar to their counterparts in C libraries. The bodies of the functions can be conditionally implemented through CUDA or OpenCL so that the same compiler translation can be reused across different lower-level accelerator APIs.

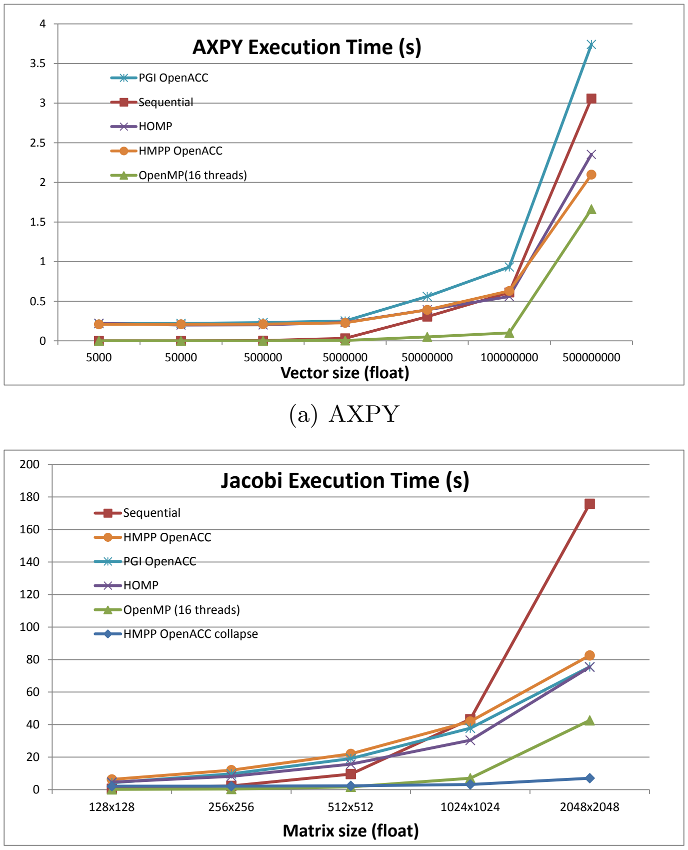
## 4 Preliminary Results

We have chosen three scientific kernels, including AXPY, Jacobi and matrix mulplication to evaluate our initial implementation. We also use the PGI [5] and HMPP [6] OpenACC compilers for comparison. All execution time measurements include the data transfer time between CPU memory and GPU memory. In addition, both sequential and OpenMP versions' performance results on CPUs are provided as a baseline.

The machine used for this evaluation has 4 quad-core Intel Xeon processors (16 cores in total) running at 2.27GHz with 32GB DRAM. An NVIDIA Tesla K20c of the Kepler architecture is installed on the machine, with CUDA version 5.0/5.0 driver as its software environment. The PGI OpenACC compiler used is version 13.4 with the command line options *pgcc -acc -ta=nvidia -Minfo=accel -mp -O3*; and the HMPP OpenACC compiler used is version 3.3.3 with command line options as *hmpp gcc -fopenmp -O3*. As source-to-source compilers, both the HOMP compiler and the HMPP compiler use GCC 4.4.7 and the CUDA 5.0 compiler as backend compilers.

Figure 5(a) shows the performance results for AXPY. The performance for the HOMP and HMPP versions are close. However, the actual computation of AXPY is very small compared to the data transfer cost, which accounts for 99% of the total execution time. Therefore, the OpenMP version of AXPY outperformed all three GPU versions when the vector size is large. The performance of using the PGI OpenACC compiler is relatively poor for large input data. We were able to look at the intermediate files generated by the PGI compiler, and have observed that the PGI compiler performs aggressive loop unrolling, which introduces a large number of branch instructions. Those instructions create divergence during thread executions, which can hurt GPU performance.

Figure 5(b) shows the performance results of Jacobi, which is computation intensive. More than 95% of the total GPU-related execution time is spent on kernel execution. While we are still working to implement collapse in HOMP, we tried to test the OpenACC version with the collapse clause, which is supported by the HMPP compiler. The PGI compiler could not compile the code when collapse is used with reduction. Without the loop collapse, the difference in performance between the three compilers is small. The use of collapse significantly improves performance on the GPU since iterations of both loops are exposed to exploit the abundant GPU threads. According to the generated CUDA code, HMPP



(a) AXPY

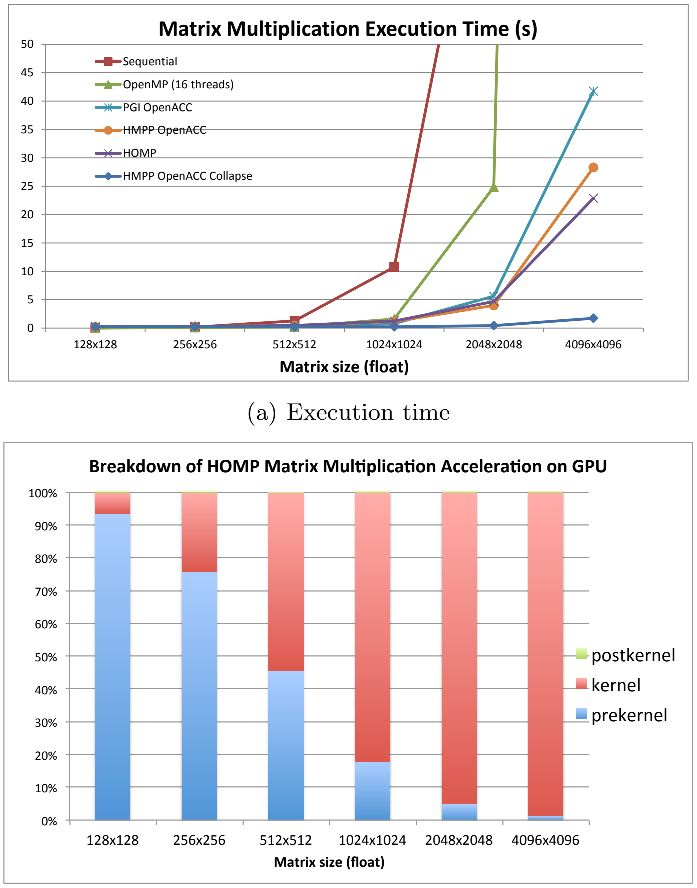
(b) Jacobi

**Fig. 5.** Performance results for AXPY and Jacobi

translates the collapse clause by mapping the associated two-level loop nest to a 2-D grid, instead of linearizing the loops.

Figure 6(a) shows results for matrix multiplication, which has significant computation for each element. Using the GPU for larger data sets easily outperforms the corresponding OpenMP version. The kernel execution time begins to dominate the total acceleration time when the data size is larger, as shown in Figure 6(b). Again, we tried to add the collapse clause for the OpenACC version of the kernel. The HMPP compiler could efficiently exploit this addition and generated much more efficient code as shown in the figure. The PGI compiler did not generate any better performance so we do not show those results.

We further compared the performance of the best generated CUDA code so far with a handwritten CUDA SDK versions and the CUBLAS version. The CUDA SDK and CUBLAS versions use completely different algorithms and apply agressive optimizations to the algorithms [7], including the use of shared memory within blocks and apply tiling to the algorithms. The performance differ-



(a) Execution time

(b) HOMP execution time breakdown

**Fig. 6.** Performance results for matrix multiplication

ence (ratio between execution times) can be large (3.21 X-9931.29 X) as shown in Table 1, although the difference decreases for larger inputs. Generation of those highly-optimized codes using a compiler is challenging without introducing new language constructs.

## 5 Discussion

Unifying programming for both CPUs and accelerators in a single high-level programming interface is an important and challenging effort. Based on our early experiences shown above, the current OpenMP extensions for accelerators are a useful step that can lead to a complete solution. The extensions are mostly intuitive for users and straightforward for compiler developers to implement. Nevertheless, the following refinements and additions would improve the usability of the OpenMP accelerator model.

| **Version/Size** | **128x128** | **256x256** | **512x512** | **1024x1024** | **2048x2048** | **4096x4096** |
|---|---|---|---|---|---|---|
| HMPP collapse | 0.238351 | 0.222798 | 0.226316 | 0.238678 | 0.444121 | 1.728459 |
| CUDA SDK | 0.000034 | 0.000141 | 0.001069 | 0.008441 | 0.067459 | 0.538174 |
| CUBLAS | 0.000024 | 0.000054 | 0.000207 | 0.001092 | 0.007229 | 0.052283 |
| Ratio(HMPP/SDK) | 7010.32 | 1580.13 | 211.71 | 28.28 | 6.58 | 3.21 |
| Ratio(HMPP/BLAS) | 9931.29 | 4125.89 | 1093.31 | 218.57 | 61.44 | 33.06 |

**Table 1.** Compare performance results of matrix multiplication (in seconds)

*Multiple Device Support* Specifying a device ID in the device() clause may not be portable. The current design may require manual code assignment and data decomposition for each device ID if multiple devices are used. New clauses such as device_type(), num_devices() and data_distribute() would support automatic code assignment and data distribution by the compiler across multiple devices.

*Combined Constructs* Separate target and parallel constructs do not intuitively express what users often want: immediate parallelism on accelerators without any sequential execution. Combined constructs such as target parallel (or target teams parallel) would conveniently meet user needs and simplify compiler implementation. Similarly, a combined teams distribute parallel for construct could be allowed so that a compiler could automatically schedule an affected loop over multiple threads from multiple teams without loop splitting in the source code.

*No-Middle-Sync Clause* Compilers may not have sufficient analysis to determine if a parallel region within a target region will have synchronization points during its execution. An implementation may have to execute the parallel region conservatively within a single CUDA thread block, which may severely under-utilize abundant GPU threads. Manually adding teams and distribute by users is often cumbersome and may not be portable. We suggest to introduce new clauses such as no-middle-sync or ignore-middle-sync to facilitate an implementation to leverage threads across multiple thread blocks. no-middle-sync expresses the semantics of no middle synchronization points while ignore-middle-sync is used to tell an implementation to ignore any possible middle synchronization points.

*Array Sections* Some may feel that RC2's different array section notations for C/C++ ([*lower-bound* : *length*]) and Fortran (built-in triplet format) are confusing. Although RC2's notation may prove more natural to C/C++ programmers, a consistent notation for both languages would fit well with many compilers that have a single IR shared by multiple languages.

*Global Barrier* The current device constructs do not support specifying synchronizations across multiple thread teams, or a league, which may often be needed. Instead, multiple target regions can be used effectively to provide barriers within a single target region. A clause (such as league or team) in the barrier directive could explicitly set the synchronization scope.

*Mapping Nested Loops* RC2 keeps the original collapse semantics, which combines multiple associated loops into one large iteration space. OpenACC has a similar clause, but does not restrict only to linearization. For example, the associated loops could be mapped to multi-dimensional grids and thread blocks when using NVIDIA GPUs. OpenMP should explicitly allow similar flexibility since linearization may not always lead to optimal performance.

*Mapped Data Reuse* In RC2, reusing mapped data relies on looking up enclosing target data regions and the data declared in the global scope using the declare target directive. Passing mapped variable pointers across a function scope may become tricky and inconvenient. A possible solution is to have explicit *liveness* attributes of mapped variables (keep, present, and final) in the map clause, a similar approach adopted in OpenACC. Making data reuse explicit can also simplify the implementation so that less runtime support is needed.

## 6 Related Work

Several previous studies [1, 6, 8–11] have explored directive-based language extensions and compiler techniques to exploit parallelism using NVIDIA GPUs. We briefly mention a few of them in this section.

OpenACC [1] is a standard for programming accelerators in conjunction with a host CPU, which could be a multicore platform. Similar to OpenMP, OpenACC programmers annotate a sequential program written in either C/C++ or Fortran with OpenACC constructs so compilers can transform the annotated program region to be executed on accelerator devices. OpenACC supports both implicit (using acc kernels) and explicit (using acc parallel) parallelism. The current OpenACC standard has limited expressivity for hybrid parallelism between CPU and GPU tasks and most compiler supports do not yet address multiple accelerators. Similarly, PGI Fortran & C accelerator extensions [8] define pragma-based directives, such as acc region and acc data region, for programmers to specify regions of computation and data to be offloaded to GPUs. An accelerator loop directive (acc for) is also provided to allow programmers to specify more explicit information for parallelizing loops. For loops without explicit scheduling clauses, its implementation relies on sophisticated compiler analysis to choose a better mapping among a few choices [5, 8].

Lee and Eigenmann [10] presented an approach of directly translating OpenMP CPU code to GPU code without using language extensions. Compiler analysis finds synchronization points in each parallel region, which can then be split into multiple subregions as necessary for generating multiple CUDA kernels. Mint [11] is a domain-specific language extension specialized in stencil kernels. Based on ROSE, Mint translates annotated C code into CUDA code. OmpSs [12, 13] is another interesting effort that allows users to define data dependences among tasks. The solution includes a powerful runtime that manages data and schedules tasks among different types of hardware devices, thus requiring little compiler support. More recently, Lee et. al. [14] compared six different directive-based GPU programming models.

Compared to previous work, our work examines OpenMP accelerator extensions and creates a prototype implementation for them. We are interested in accelerator language extensions that are compatible with existing OpenMP execution and memory models. Consequently, the implementation techniques that we explore are based on an existing open-source OpenMP compiler.

## 7 Conclusions and Future Work

In this paper, we have examined the newly introduced accelerator model in OpenMP 4.0 (RC2) and shared our experiences of creating a prototype implementation for it. Our implementation has already been released under a BSD license as part of the ROSE compiler framework.

The OpenMP accelerator extensions represent a major enhancement for OpenMP to meet the increasing demands to support accelerators and heterogeneous architectures. For developers, most extensions are intuitive and fit well with OpenMP's existing execution model and memory model. Complexity arises from the use of teams and distribute constructs to organize the thread teams and hierarchy. Combined constructs are needed. For compiler developers, creating a working implementation that leverages an existing OpenMP compiler framework is straightforward based on our early experience, though aggressive compiler analysis and optimization techniques further enhance the performance of generated codes. It also requires efficient runtime support to manage the data mapping and to coordinate the executions of CPU tasks and accelerator kernels.

Our future work includes the following research directions. We will target more hardware architectures such as the Intel Many Integrated Core Architecture (MIC). We also will generate OpenCL in addition to CUDA. We will investigate techniques to aid users in choosing between CPU threads, accelerators and vectorization; We will explore a peer-to-peer execution model that can express code and data offload without always involving a host device.

## References

1. "OpenACC: Directives for Accelerators," http://www.openacc-standard.org/.
2. OpenMP Architecture Review Board, "The OpenMP API Specification for Parallel Programming," http://www.openmp.org/.
3. C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski, "A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries," in *IWOMP*, ser. Lecture Notes in Computer Science, M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, and B. R. de Supinski, Eds., vol. 6132. Springer, 2010, pp. 15–28.
4. D. Quinlan *et al.*, "ROSE Compiler Infrastructure," http://www.rosecompiler.org/.
5. M. Wolfe, "Implementing the PGI Accelerator Model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 43–50.
6. R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A Hybrid Multicore Parallel Programming Environment," 2007.
7. V. Volkov and J. W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 31:1–31:11.

8. The Portland Group, “PGI Fortran & C Accelerator Compilers and Programming Model,” Tech. Rep., November 2008.
9. T. D. Han and T. S. Abdelrahman, “hiCUDA: A High-Level Directive-Based Language for GPU Programming,” in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 52–61.
10. S. Lee and R. Eigenmann, “OpenMPC: Extended OpenMP Programming and Tuning for GPUs,” in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.
11. D. Unat, X. Cai, and S. B. Baden, “Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C,” in *Proceedings of the international conference on Supercomputing*, ser. ICS ’11. New York, NY, USA: ACM, 2011, pp. 214–224.
12. A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, “OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures,” *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.
13. J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta, “Productive Programming of GPU Clusters with OmpSs,” in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 557–568.
14. S. Lee and J. S. Vetter, “Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 23:1–23:11.